

Overfitting and Remedies

Find the perfect model complexity, Early stopping, Regularization, Dropout, Data augmentation, and Domain randomization

Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - Find the perfect model complexity
 - Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Recap: Parameter and Gradient Values

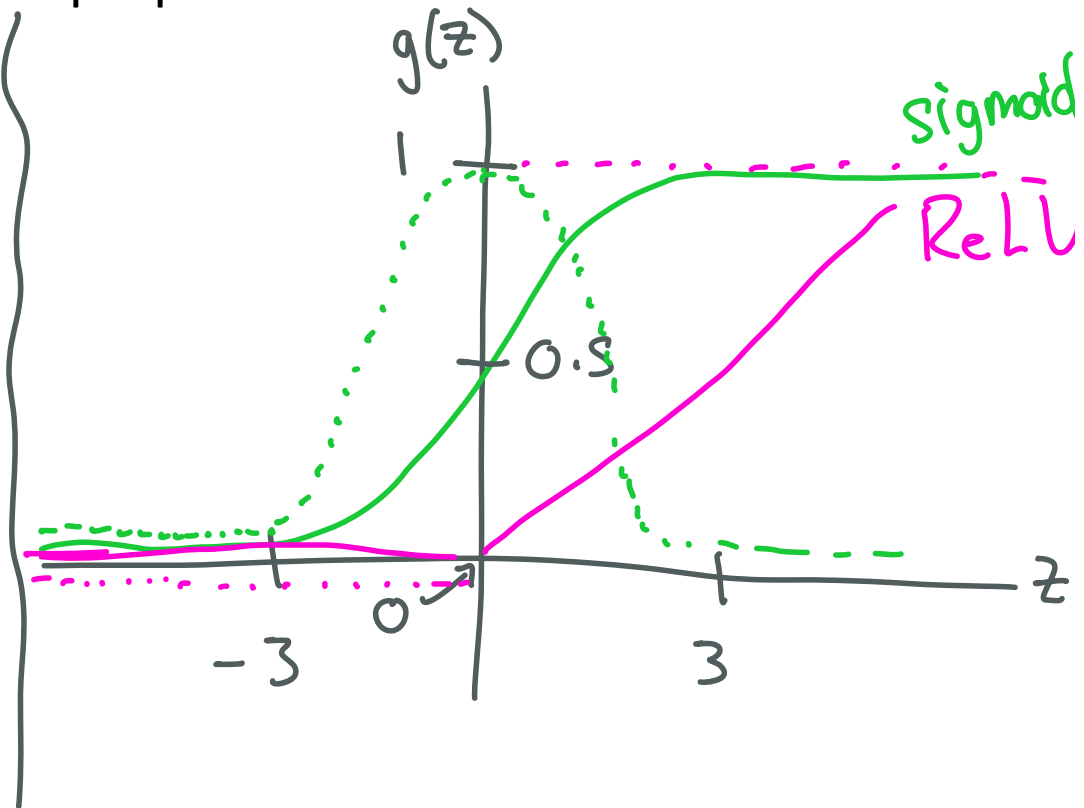
- Take five minutes to draw
- Example: activations with and without proper initialization and normalization



$$z = A^{[l-1]} w^T + b$$

$$A^{[l]} = g(z)$$

activation
function



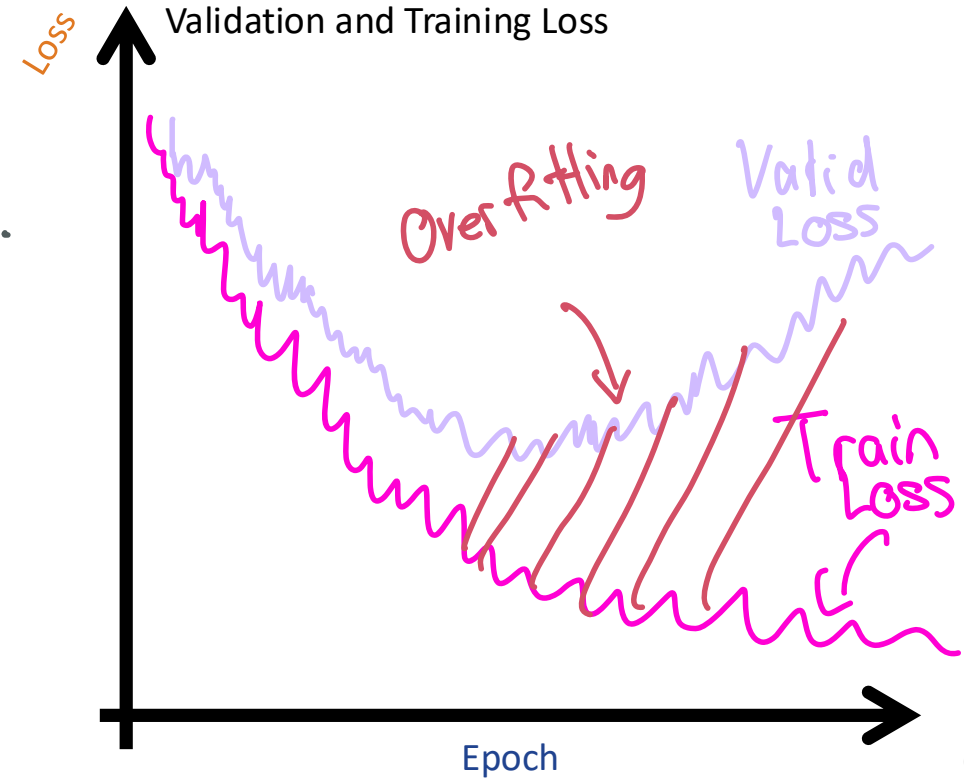
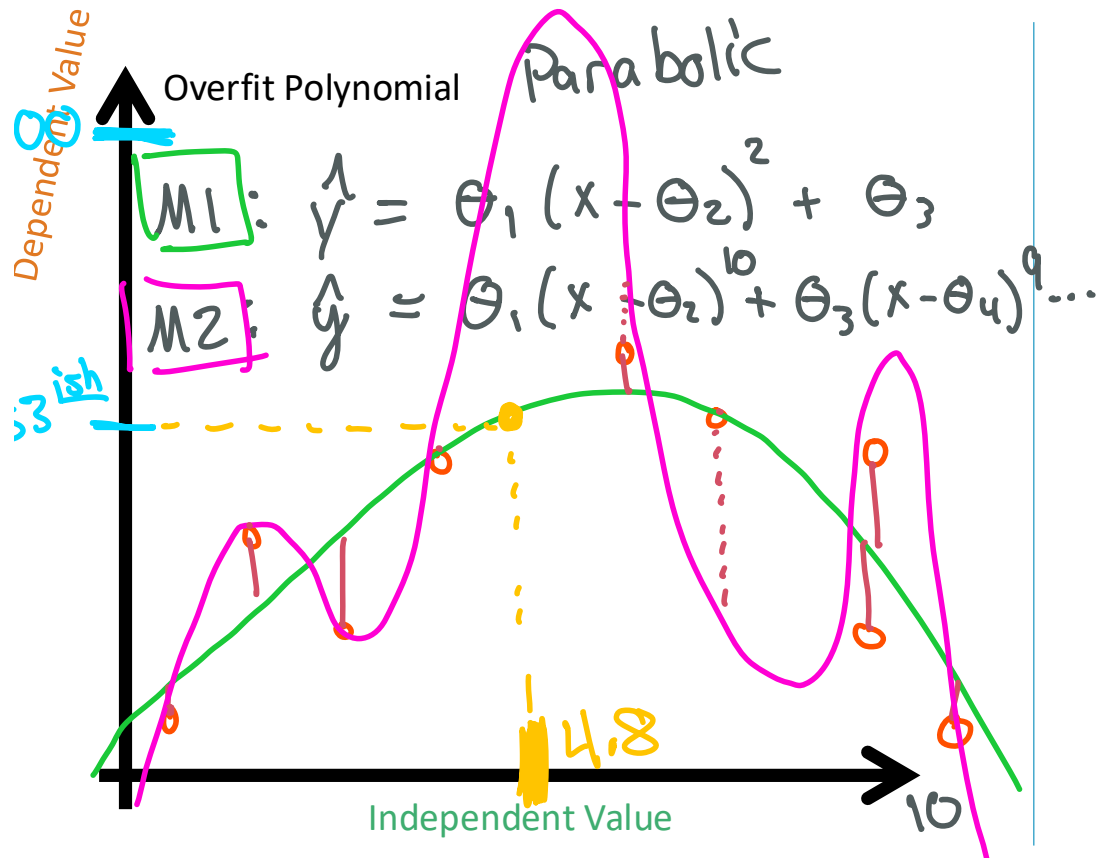
Classroom Etiquette

- We all want to look *effortlessly smart* in front of our peers.
 - It's a fool's errand. I've noticed it a bit in the class. Might be due to class makeup
- I've built my teaching philosophy around the “gift of failure”
 - You need to give me wrong answers
 - You need to be unafraid of being wrong
 - You need to be ready to fail

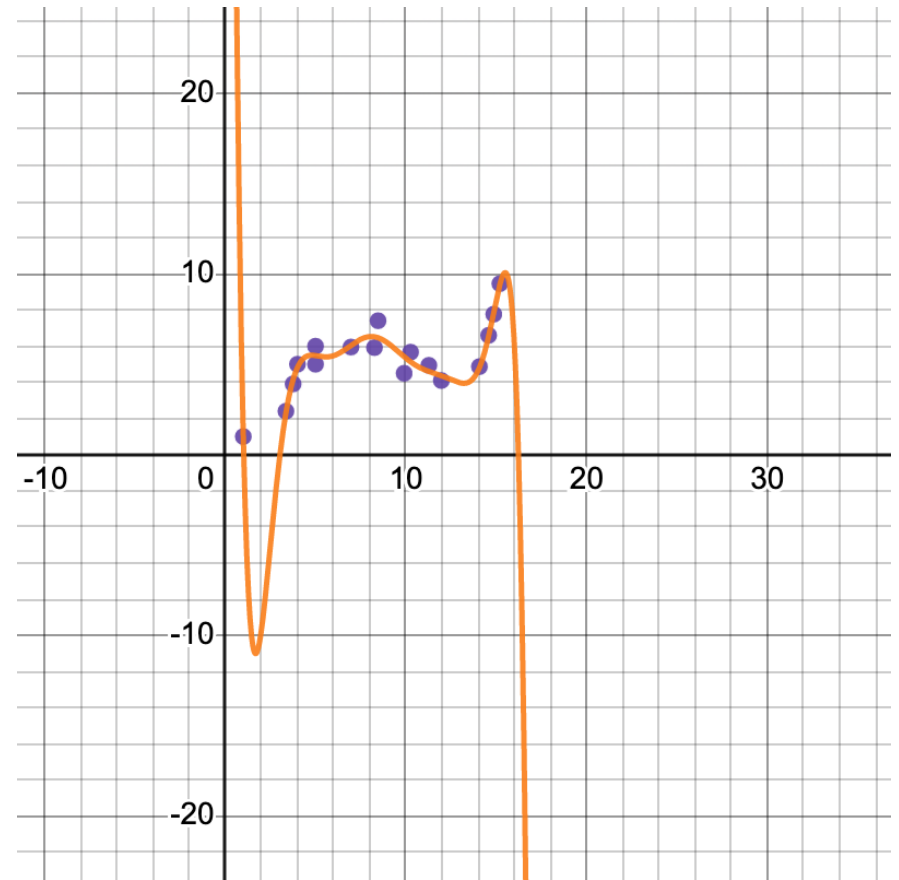
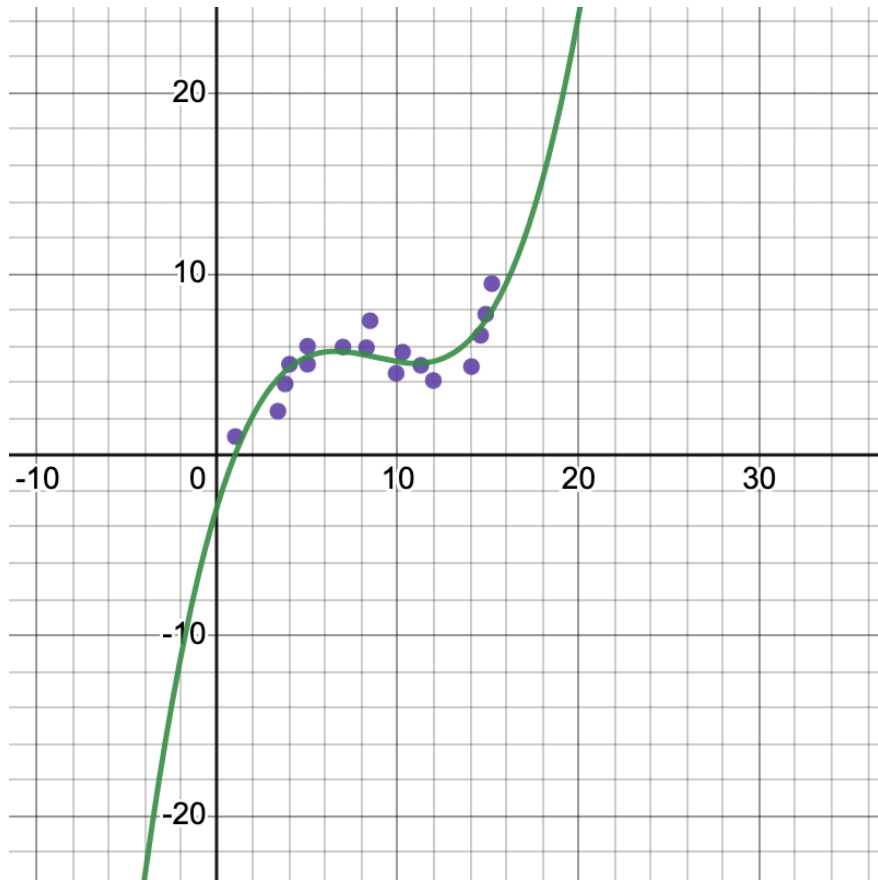
$$\frac{1}{2} (\hat{y} - y)^2 \quad \left\{ \quad |\hat{y} - y| \right.$$

Overfitting

When your model **learns/memorizes** the training data and not **some property** that is useful for inference. ("I've seen this input before... the answer is X.")



<https://www.desmos.com/calculator/gysbxd1r0l>



Causes of Overfitting

When your model **learns/memorizes** the training data and not **some property** that is useful for inference. (*"I've seen this input before... the answer is X."*)

- The model is too complex
 - Too many parameters
 - Too deep
 - Too wide
 - Too much memory

Causes of Overfitting

When your model **learns/memorizes** the training data and not **some property** that is useful for inference. (*"I've seen this input before... the answer is X."*)

- The model is too complex

- Too many parameters
- Too deep
- Too wide
- Too much memory

80, 100

$$\hat{y} = \theta_1 x^{10} + \theta_2 x^9 \dots + \theta$$

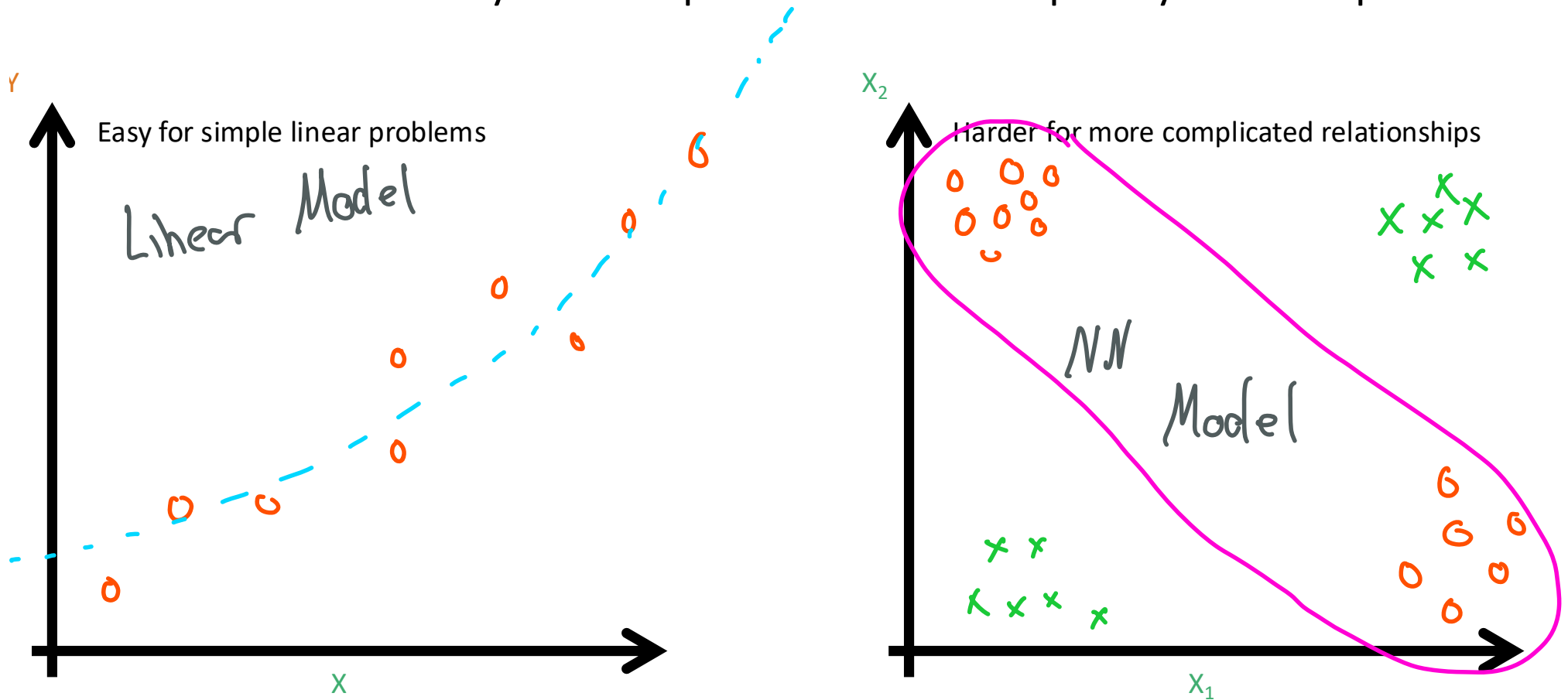
- Parameters are too large (large parameters lead to steep curves)
- The model was trained for too long
- The dataset was too small

Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - ✓ • Find the perfect model complexity
 - ✓ • Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Remedy: Find the Perfect Model Complexity

We could theoretically find the perfect model complexity for each problem



Hyperparameter Search/Tuning

- Common methods for “finding” good hyperparameters include
 - Manual adjustments
 - Grid search
 - Random search
 - Bayesian optimization
 - Evolutionary optimization
 - (and others)
- I happen to prefer a simple “Twiddle Search”

```

# Initial values
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 64,
    "num_layers": 10,
    "dropout": 0.5,
}

# Hyperparameter update factors
hyper_param_updates = {
    "learning_rate": {"up": lambda lr: lr * 10, "down": lambda lr: lr / 10},
    "batch_size": {"up": lambda bs: bs * 2, "down": lambda bs: max(bs // 2, 1)},
    "num_layers": {"up": lambda nl: nl * 2, "down": lambda nl: max(nl // 2, 1)},
    "dropout": {"up": lambda d: min(d + 0.1, 0.9), "down": lambda d: max(d - 0.1, 0.1)},
}

# Initial quality
best_metric_value = evaluate(hyper_params)

# Cache of hyperparameter value combinations
cache = {hyper_params.values(): best_metric_value}

attempts = 1
while not done(best_metric_value, attempts):

    # Choose a hyperparameter and an update direction
    hyper_param = choice(list(hyper_params.keys()))
    update_direction = choice(["up", "down"])

    # Update the hyperparameter
    current_value = hyper_params[hyper_param]
    new_value = hyper_param_updates[hyper_param][update_direction](current_value)
    new_hyper_params = {**hyper_params, hyper_param: new_value}

    # Check if the hyperparameter value combination has been evaluated before
    if new_hyper_params.values() in cache:
        continue

    attempts += 1

    # Evaluate the new hyperparameter value combination
    metric_value = evaluate(new_hyper_params)
    cache[new_hyper_params.values()] = metric_value

    if metric_value > best_metric_value:
        best_metric_value = metric_value
        hyper_params = new_hyper_params

print(f"Best metric value: {best_metric_value}: {hyper_params}")

```

Initial values

```
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 64,
    "num_layers": 10,
    "dropout": 0.5,
}
```

} 4

$$4 \cdot 5 = 20$$

Hyperparameter update factors

```
hyper_param_updates = {
    "learning_rate": {"up": lambda lr: lr * 10, "down": lambda lr: lr / 10},
    "batch_size": {"up": lambda bs: bs * 2, "down": lambda bs: max(bs // 2, 1)},
    "num_layers": {"up": lambda nl: nl * 2, "down": lambda nl: max(nl // 2, 1)},
    "dropout": {"up": lambda d: min(d + 0.1, 0.9), "down": lambda d: max(d - 0.1, 0.1)},
}
```

```
# This function updates the hyperparameters of the model
def update_hyperparams(model, hyper_params, hyper_param_updates):
    # Check if the hyperparameters are valid
    for key, value in hyper_params.items():
        if not isinstance(value, (int, float, str)):
            raise ValueError(f"Hyperparameter {key} must be of type int, float, or str")

    # Check if the hyperparameters are within the allowed range
    for key, value in hyper_params.items():
        if key == "learning_rate" and value < 0.001 or value > 1.0:
            raise ValueError(f"Learning rate must be between 0.001 and 1.0")
        elif key == "batch_size" and value < 1 or value > 1000:
            raise ValueError(f"Batch size must be between 1 and 1000")
        elif key == "num_layers" and value < 1 or value > 100:
            raise ValueError(f"Number of layers must be between 1 and 100")
        elif key == "dropout" and value < 0.0 or value > 1.0:
            raise ValueError(f"Dropout rate must be between 0.0 and 1.0")

    # Update the hyperparameters
    for key, value in hyper_params.items():
        if key in hyper_param_updates:
            for update in hyper_param_updates[key]:
                new_value = update(value)
                hyper_params[key] = new_value

    # Print the updated hyperparameters
    print("Updated hyperparameters:", hyper_params)
```

```
# Initial quality
best_metric_value = evaluate(hyper_params)

# Cache of hyperparameter value combinations
cache = {hyper_params.values(): best_metric_value}

attempts = 1
while not done(best_metric_value, attempts):
```

1!

```

# This file is a wrapper
hyper_params = {
    "learning_rate": 0.1,
    "max_epochs": 10,
    "min_epochs": 0.1,
}

# Hyperparameter update function
def hyper_params_update(params):
    """Update hyperparameters based on the current value of the parameter"""
    # Update learning rate
    lr = params["learning_rate"]
    if lr > 0.1:
        lr = 0.1
    elif lr < 0.01:
        lr = 0.01
    params["learning_rate"] = lr

    # Update max epochs
    max_epochs = params["max_epochs"]
    if max_epochs > 10:
        max_epochs = 10
    elif max_epochs < 0.1:
        max_epochs = 0.1
    params["max_epochs"] = max_epochs

    # Update min epochs
    min_epochs = params["min_epochs"]
    if min_epochs > 10:
        min_epochs = 10
    elif min_epochs < 0.1:
        min_epochs = 0.1
    params["min_epochs"] = min_epochs

    return params

# This file is a wrapper
def eval_model(model, params):
    """Evaluate the model with the given hyperparameters"""
    # Evaluate the model
    results = {}
    for param in params.keys():
        results[param] = eval_model(model, params[param])

    return results

# This file is a wrapper
def train_model(model, params):
    """Train the model with the given hyperparameters"""
    # Train the model
    results = {}
    for param in params.keys():
        results[param] = train_model(model, params[param])

    return results

# This file is a wrapper
def test_model(model, params):
    """Test the model with the given hyperparameters"""
    # Test the model
    results = {}
    for param in params.keys():
        results[param] = test_model(model, params[param])

    return results

```

Choose a hyperparameter and an update direction

hyper_param = choice(list(hyper_params.keys()))

update_direction = choice(["up", "down"])

Update the hyperparameter

current_value = hyper_params[hyper_param]

new_value = hyper_param_updates[hyper_param][update_direction](current_value)

new_hyper_params = {**hyper_params, hyper_param: new_value}

Check if the hyperparameter value combination has been evaluated before

if new_hyper_params.values() **in** cache:

continue

```

attempts += 1

# This file is a wrapper
def eval_model(model, params):
    """Evaluate the model with the given hyperparameters"""
    # Evaluate the model
    results = {}
    for param in params.keys():
        results[param] = eval_model(model, params[param])

    return results

# This file is a wrapper
def train_model(model, params):
    """Train the model with the given hyperparameters"""
    # Train the model
    results = {}
    for param in params.keys():
        results[param] = train_model(model, params[param])

    return results

# This file is a wrapper
def test_model(model, params):
    """Test the model with the given hyperparameters"""
    # Test the model
    results = {}
    for param in params.keys():
        results[param] = test_model(model, params[param])

    return results

```

```

# This line is a placeholder
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 32,
    "num_epochs": 10,
}

# Hyperparameter search space
hyper_params_search_space = {
    "learning_rate": ("log", lambda l: l in [0.01, 0.1, 1]),
    "batch_size": ("log", lambda b: b in [8, 16, 32, 64]),
    "num_epochs": ("log", lambda e: e in [10, 20, 50, 100])
}

# This line is a placeholder
best_hyper_params = None

# This line is a placeholder
best_metric_value = None

# This line is a placeholder
cache = {}

# This line is a placeholder
attempts = 1

# This line is a placeholder
hyper_params = hyper_params_search_space.sample()

# This line is a placeholder
metric_value = evaluate(hyper_params)

# This line is a placeholder
cache[hyper_params] = metric_value

# This line is a placeholder
if metric_value > best_metric_value:
    best_metric_value = metric_value
    hyper_params = hyper_params

# This line is a placeholder
print(f"Best metric value: {best_metric_value}: {hyper_params}")

```

attempts += 1

Evaluate the new hyperparameter value combination

```
metric_value = evaluate(new_hyper_params)
cache[new_hyper_params.values()] = metric_value
```

```
if metric_value > best_metric_value:
    best_metric_value = metric_value
    hyper_params = new_hyper_params
```

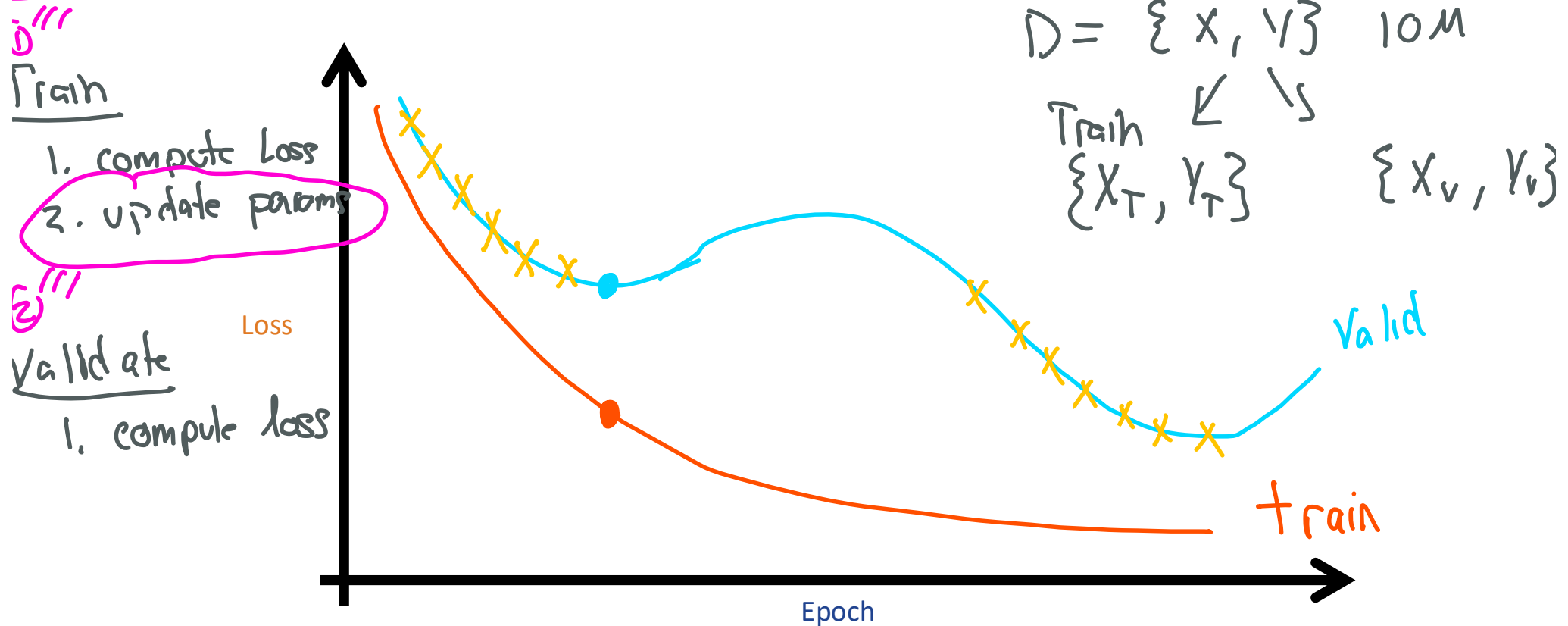
```
print(f"Best metric value: {best_metric_value}: {hyper_params}")
```


Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - Find the perfect model complexity
 - Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Remedy: Early Stopping and Checkpointing

~~002~~ We can use the learned parameters from before we detected overfitting



Checkpointing

```
for epoch in range(num_epochs):
    model.train()
    for X, y in train_loader:
        yhat = model(X)
        loss = criterion(y, yhat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        for X, y in valid_loader:
            yhat = model(X)
            loss = criterion(y, yhat)
            metric = metrics(y, yhat, model, metric)

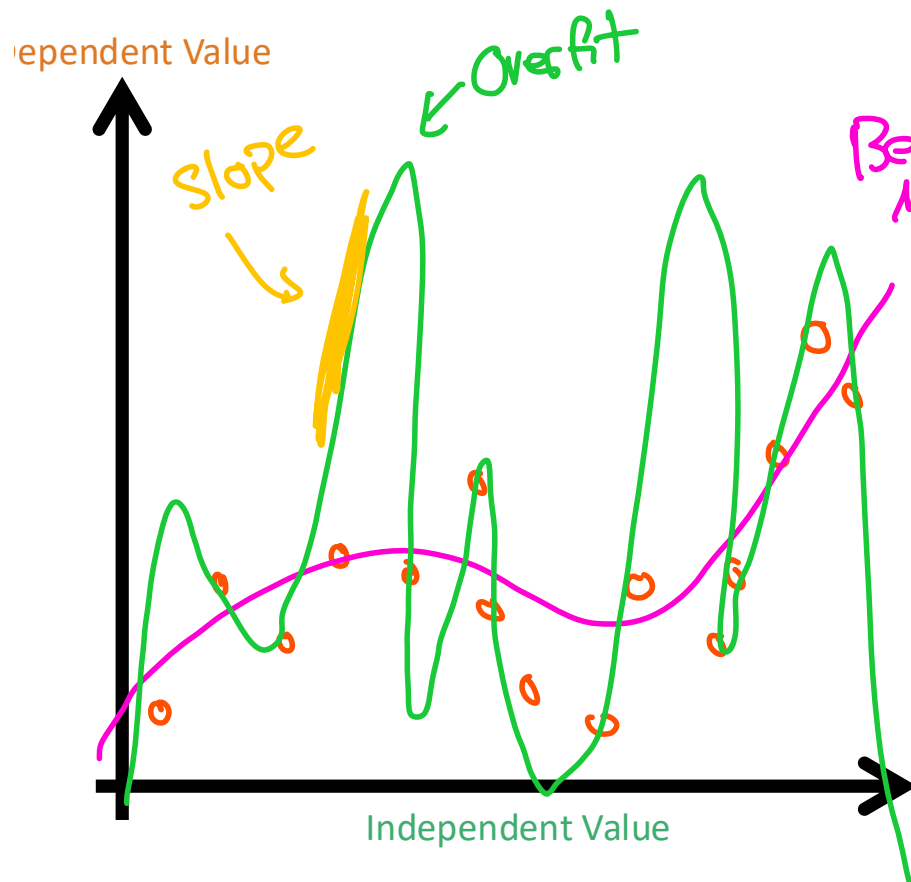
    if metric.is_best():
        model.save(f"model{epoch}.pkl")
```

Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - Find the perfect model complexity
 - Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Remedy: Regularization

We can artificially *constrain* the parameter magnitudes in our loss function
(ie, optimize for lower parameter magnitudes)



$$\mathcal{L} = \frac{1}{2} (\hat{y} - y)^2$$

How would you change this function so that it is also minimizes parameter values.

$$\mathcal{L} = \frac{1}{2} (\hat{y} - y)^2 + \lambda \frac{1}{2} \|\Theta\|^2$$

weight decay

Derivative of $\frac{1}{2}$ MSE with Regularization

$$J = \frac{1}{2}(\hat{y} - y)^2 + \lambda \frac{1}{2} \|\Theta\|^2$$

$$\frac{\partial J}{\partial \Theta} = (\hat{y} - y) \hat{y}' + \lambda \Theta$$

$$\Theta_t := \Theta_{t-1} - \alpha [(\hat{y} - y) \hat{y}' + \lambda \Theta]$$

(1) Θ is a large positive value

↳ reduce Θ

(2) Θ is a large negative value

↳ increase Θ

tend
toward
zero

$$J = \frac{1}{2}(\hat{y} - y)^2 + \Theta$$

$$\frac{\partial J}{\partial \Theta} = (\hat{y} - y) \hat{y}' + 1$$

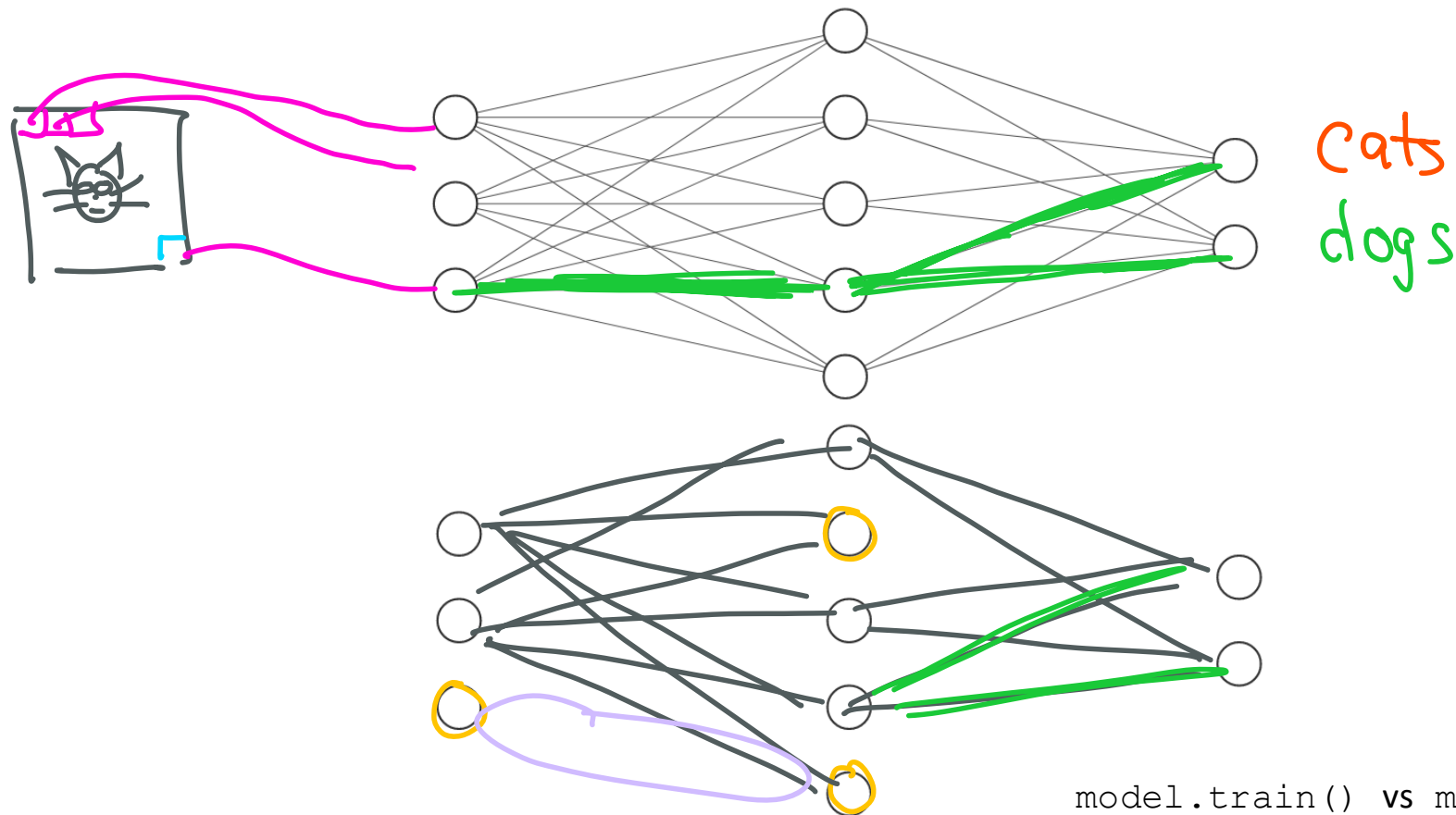
$$\Theta_t := \Theta_{t-1} - \alpha ((\hat{y} - y) \hat{y}' + 1)$$

Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - Find the perfect model complexity
 - Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Remedy: Dropout

We can train the model in such a way that breaks memorization



Remedy: Dropout

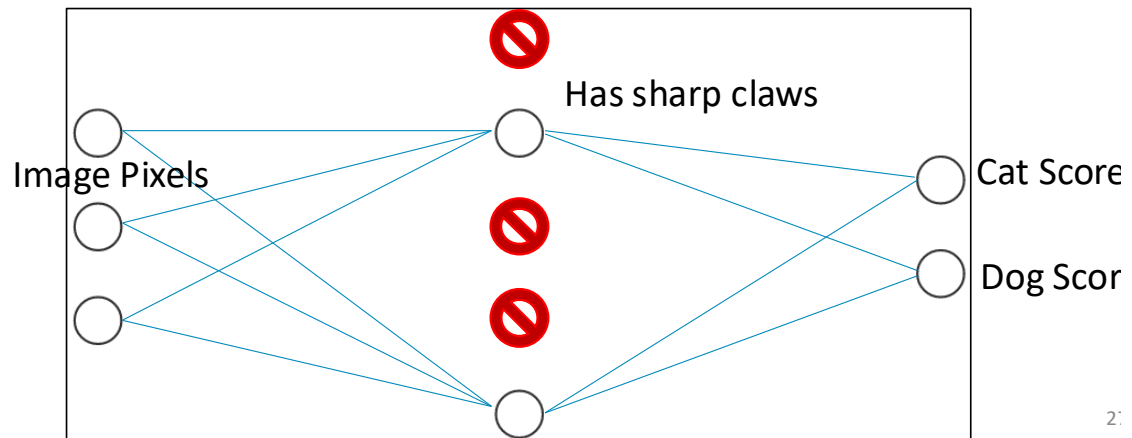
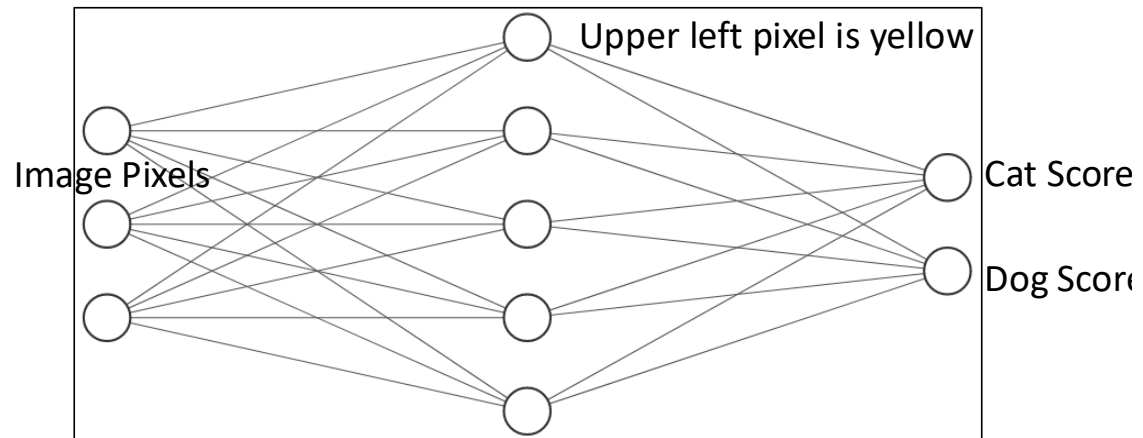
We can train the model in such a way that breaks memorization

$$z = A^{[l-1]} w^T + b$$

- Randomly set neuron outputs to zero
- Choose a different set of neurons each time
- The model needs redundant representations
- This leads to more general representations
- A single pathway cannot memorize the input

```
In model.train() mode
for layer in model.layers():
    keep_prob = 1 - dropout_rate
    keep = torch.rand_like(layer.shape) < keep_prob
    activation *= keep.float()
    activation /= keep_prob
```

```
In model.eval() mode
for layer in model.layers():
    activation *= 1.0
```

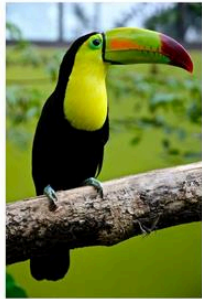


Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - Find the perfect model complexity
 - Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Remedy: Data Augmentation

<https://albumentations.ai>



Original



Mirrored



Rotated



Brighter



Original image

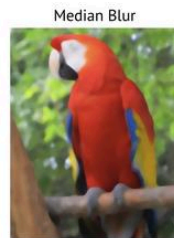
augmentation →



Horizontal Flip



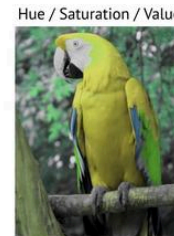
Crop



Median Blur



Contrast



Hue / Saturation / Value



Gamma

```
for epoch in range(num_epochs):  
    model.train()  
    for X, y in train_loader:  
        yhat = model(X)  
        loss = criterion(y, yhat)  
        optimizer.zero_grad()  
        loss.backward()  
        optimizer.step()
```

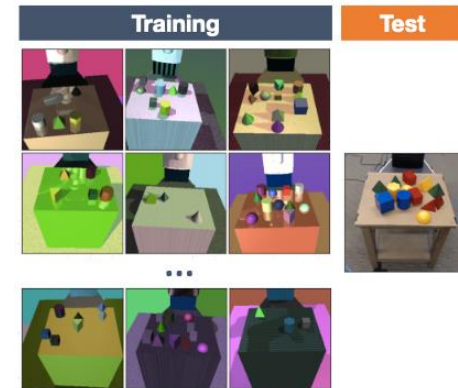
```
model.eval()  
with torch.no_grad():  
    for X, y in valid_loader:  
        yhat = model(X)  
        loss = criterion(y, yhat)  
        metric = metrics(y, yhat, model)
```

Outline

- Drawing recap for initialization and normalization
- Overfitting and its causes
- Overfitting remedies
 - Find the perfect model complexity
 - Early stopping
 - Regularization
 - Dropout
 - Data augmentation
 - Domain randomization

Remedy: Domain Randomization

- This process happens during the data synthesis/creation process.
- It often relies on simulation, and it is frequently used to cross the simulation-to-reality gap.
- This is often called Sim2Real in machine learning and robotics.



“Illustration of our approach. An object detector is trained on hundreds of thousands of low-fidelity rendered images with random camera positions, lighting conditions, object positions, and non-realistic textures. At test time, the same detector is used in the real world with no additional training.”

— [Tobin et al](#)

Summary

- Models can accidentally memorize the input data instead of learning some useful, general property
- We can prevent overfitting/memorization with several remedies
- Most remedies try to
 - Artificially limit the magnitude of parameter values (early stopping, regularization)
 - Add noise and randomness to the training process (dropout, augmentation, domain randomization)
- We often use these remedies together